Postel 5708

Network Working Group
Request for Comments: 91

George H. Mealy
Harvard University
December 27, 1970

A PROPOSED USER-USER PROTOCOL

INTRODUCTION:

There are many good reasons, and maybe one or two bad ones, for making it appear that communication over the Network is only a special case of input/output -- at least as far as user programming is concerned. Thus, for instance, the Harvard approach toward implementing the HOST-HOST protocol and Network Control Program treats each link as a "logical device" in PDP-10 terminology. Setting up a connection is similar to local device assignment, and communication over a link will make use of the standard system input/output UUO's. This makes it possible to use existing programs in conjunction with the Network without modification -- at least if other PDP-10's are being dealt with.

This takes us only so far, however. The notion of a "logical record" does not exist on the PDP-10; it does on the IBM 360 (I am speaking here at the level of the operating system -- user program interface). Furthermore, in the absence of a Network standard requiring fixed representations for integers, reals, etc. (which I would oppose), any pair of user processes must arrive at a local agreement, and one or both must assume the burden of data conversion where necessary. Any standard

protocol should allow such agreements to be given expression and should accommodate at least the minimum of control information that will allow such agreements to function in practice. Finally, we must note that the IMP-IMP and HOST-HOST protocols do not provide for a check that an action requested by a user process is actually accomplished by the other process; this type of issue has always been regarded as subject to treatment at the USER-USER protocol level.

This proposal is intended to face the above three types of issue only to a certain extent. I can best explain that extent by stating the criteria I would use to judge any USER-USER protocol proposal:

1. The notion of a (logical) record should be present, and the notion of a message should be suppressed. (To a FORTRAN programmer, that which is written using one WRITE statement with no accompanying FORMAT is a record; to an OS/360 machine language programmer, PUT writes a record).

2. It should be possible to so implement the protocol in HOST systems and/or library routines that now existing user programs can access files anywhere in the Network without program modification. (Initially, at least, this ability must be restricted to HOST systems of the same type).

3. The protocol should be implementable (not necessarily implemented) in any HOST system at the SVC or UUO

level. Specific knowledge of the characteristics of the other HOST involved should be unnecessary.

It should be noted that the above imply that some user programs must be aware of the nature of the other HOST -- at least in each case where the second criterion fails. As we make progress in (or give up on) the cases where the failure now occurs, the burden of accommodating system differences will shift toward implementation in protocols (i. e. , the HOST systems) or, by default, in user programs.

Quite clearly, any proposal initiated today should be suspect as to the extent to which it "solves" ultimate problems. How ambitious to be is strictly a matter of taste. At this stage, I prefer to try something which I believe can be used by all of us (and, hence, is worth doing), goes a reasonable distance towards solving our short-range problems, is easy to do, and offers hope of viability in the long range view. In the following, I intend to describe the proposal itself with, I hope, proper motivational arguments for its pieces. I will then sketch the specific implementation we at Harvard are making for the PDP-10 and describe how we intend to apply it in the specific case of storage of files on other PDP-10's in the Network.

USER-USER PROTOCOL (PROPOSAL)

The following protocol is intended to apply to the data bits in messages between the end of the marking bits and the beginning of the padding bits. The present IMP-IMP and HOST-HOST protocols are unaffected by this proposal.

The general principle is that each segment (this is not a technical term) of data is preceded by control information specifying its nature and extent. The basic scheme has been evolved from that used in the SOS buffering system (see the papers in JACM, April 1959 and especially that by O.R. Mock).

Our point of view is that a link is a carrier of information. Information is carried in segments of a fixed maximum length, called messages*. That this is so is an accident, from the user's point of view; when he wishes to transmit a contiguous stream of data, he will in general, segment it in a different (from the IMP-IMP or HOST-HOST protocol point of view) manner -- we will call his segment a record. It should be clear that this is entirely analogous to the distinction made in (sequential) file storage systems between the notion of (physical) block and (logical) record. On the side, file storage systems also make use of control and status information; we will also.

At the USER-USER protocol level, all information transmitted over the link is a sequence of flags followed by (possibly null) data blocks. The general format will be:

OPERATION COUNT DATA

The OPERATION field is always present and is four bits long. The COUNT field, when present, gives the number of data

---

*A message is that string of bits between any two HOST-HOST headers.

bytes following in the data block. The byte size is set by the last preceding SIZE flag (in most cases). The byte may be between zero and 255 bits long (Yes, Virginia, zero is zero even when you have a System/360). The OPERATION field and the COUNT field (when present) are called the flag and the data bytes (when present) the data block. Flags followed by data blocks (even when null due to a zero count) are called block flags, and other flags are called whyte* flags.

It is to be noted that, since the SIZE flag sets the byte size for the following blocks, byte size may be set at that "natural" for the sending or for the receiving HOST, depending on local agreement between the sending and receiving user processes. It is specifically required that a SIZE flag appear in each message prior to any block flag (except the ASCII flag); the SIZE flag may be introduced on a default basis by the routine(s) implementing the protocol and is intended partially as a means of detecting certain classes of error.

The COUNT field is 8 bits in length (except in the EOM flag, where it is 16 bits long). The flags are as follows:

Whyte Flags:

0 - NUL	No operation (consider next flag)
1 - RS	Record Separator (end of record)

---

*In memory of an attractive, but nonspelling, SDC secretary who could not distinguish between black and white, at least during 1957 and in manuscript form.

2 - GS	Group Separator (end of group)
3 - FS	File Separator (end of file)
4 - ESC	Escape to local convention for flags
5 -	(reserved for later assignment)
6 - EOM N	End of Message (N is total bit count)
7 - SIZE N	Byte size is N bits
8 - IGNORE N	Ignore following data bits

Block Flags:

9 - SYS N	N bytes of data for receiving HOST system
10 - CONTROL N	N bytes of control data follow
11 - STATUS N	N bytes of status data follow
12 - LABEL N	N bytes of identification data follow
13 - KEY N	N bytes of key data follow
14 - ASCII N	N (8-bit) bytes of ASCII data follow
15 - BLOCK N	N bytes of data follow

I have already mentioned the requirement for SIZE. Absence of the SIZE flag in any message containing block flags (except ASCII) is a definite error. EOM is partially another error-checking device and partially a device for bypassing the padding conundrum. A user program should never see EOM on input; the user may write an EOM to force transmission. EOM delimits the end of the useful information in the message and restates the total number of bits in the message, starting with the first bit following the marking and ending with the last bit of the EOM count field, to check possible loss of information. This is a check

against errors in the IMP-HOST electrical interface and in the HOST mushyware. EOM must appear at the end of each message, unless ESC has appeared.

ESC is intended as a (hopefully) unused ascape hatch, for nonuse by those installations and/or applications wishing to avoid using more than four bits of the USER-USER protocol on any link. For instance, it may be desired to use a link as a bit stream, ignoring even message boundaries. If and when anarchists can achieve local agreement, more power to them!

NUL and IGNORE are intended to be space fillers, in case it is helpful to make the first bit of the subsequent data block occur on a convenient address boundary. (An especially helpful HOST interrupt routine might even paste a combination of NUL and IGNORE over the marking bits when receiving a message -- in which case, their bit count should be transmitted on to the GET routines to correct the EOM bit count check). The separator operations introduce the notions of logical record, group, and file. Specifically, there is no requirement that a record be contained entirely within a message or that only a single record be contained in a message! In addition, there is no requirement that only one file be transmitted during a connection. For instance, a user might wish to use a link to transmit a collection of routines, and then do something else with the link. By local agreement, then, a single routine might consist of a number of records forming a group, the whole collection might form a file, and the link might remain connected after the FS flag is received.

The interpretation of the various block flags is similarly open to local agreement. The two flags intended to convey pure data are ASCII and BLOCK; the difference between them is only (as far as the protocol is concerned) that the byte size is implicit for ASCII (8 bits) and explicit for BLOCK (the count field of the next preceding SIZE flag). Beyond this, however, the semantic content of the block following ASCII is governed by the current standards for ASCII; EBCDIC information may not be transmitted in an ASCII block!!

CONTROL and STATUS are intended for communication of control information between user processes, and the interpretation of their accompanying data blocks is open to local agreement. Generically, CONTROL means "try to do the following" and STATUS means "but I feel this way, doctor." A CONTROL flag will prompt a returned STATUS flag, sooner or later, or never. LABEL is intended for use in identifying the following unit(s) of data, at the file or group level. Again, the specific interpretation is a matter of local agreement. KEY is intended to mimic the notion of address or key -- this is at the record, data item, or even physical storage block level. For those familiar with PDP-10 system and/or OS/360, the following parallels are offered for guidance:

USER-USER protocol	OS/360	PDP-10
CONTROL	OPEN	OPEN
	CLOSE	CLOSE
LABEL	DSCB	File retrieval information
KEY	KEY	USETI/USETO argument
CONTROL	READ	IN/INPUT
	WRITE	OUT/OUTPUT
	ALLOCATE ?	ENTER
	OPEN ?	LOOKUP
STATUS	?	GETSTS

The "?" notations above indicate lack of a very direct parallel. It is worth noting that the OS/360 GET and PUT have direct parallels in any implementation of the USER-USER protocol that embodies the notion of record; our implementation of the protocol will lead to introduction of this notion for all PDP-10 input/output involving disc and tape storage, as well as IMP communication.

If I knew the MULTICS terminology, I could extend the set of parallels above with more precision. Although my terminology has been drawn from systems with explicit input/output imperatives, I wish to emphasize that this setup in intended to handle control and data communication in general; MULTICS is a system in which the classical distinction between external and internal storage is blurred (from the user's point of view) in a manner in which I wish it blurred in the USER-USER protocol.
I offer SYS with only slight trepidation. The general notion is that one should be able to communicate directly with a foreign HOST

rather than via a foreign user process as its intermediary, SYS is like a UUO or SVC, but for the foreign HOST's consumption rather than my HOST's. From the HOST's point of view, the problem in implementation is in establishing a process context record unconnected with any local user process. This, however, is strongly associated with our current LOGON conundrum. On the PDP-10, for instance, users are more or less identified with local teletype lines, and any link is not one of those! Hence, subterfuge is necessary to let a foreign user log on. OS/360 is as (actually, more) perverse in its own way.

The process of logging a foreign process onto my local system is not (except possibly for MULTICS) a simple matter of having a special (!) user job present which is responsible for doing it. When and if anything else is possible, the HOST must provide a system instruction (UUO or SVC or whatever) that gives the requisite information establishing a process independent in all senses of the process that made the request. Otherwise, self-protection mechanisms which are reasonable for any system will make us all much more interdependent than we wish. To do this, there must exist in every system a UUO/SVC that does the right thing (ATTACH, but forget me). If this is true, then the LOGON process over the Network is tantamount to issuance of a foreign UUO/SVC by another node in the Network. I see no reasonable way around this. If that is the case, then SYS N is the kind of flag to use to convey the requisite data. If that is so, then it is only reasonable to let SYS convey a request for any OS instruction at the user program-operating system interface level!

The practical questions of implementation are something else! In the case of the PDP-10, I can pretty well see how to turn a SYS into either a LOGON request or a request to execute a monitor command or UUO (would that they were the same) as the case might be. OS/360 is more sophisticated, unfortunately. MULTICS might make it. Naytheless, I hope that it is clear that what we want to do, which is what the protocol should reflect, is quite a different question from that of how it is to be done in the context of a specific HOST system. What we want to do is, in general, rather independent of the system we are dealing with as far as the protocol is concerned, and we should not fail to introduce general notions into the protocol just because we are uncertain as to how they may have to be translated into particular implementation practice.

A PDP-10 IMPLEMENTATION

Although the following can be implemented as either a set of user routines or imbedded in the monitor as UUO's (our first implementation will be the former), the latter version will be used here for descriptive purposes. The UUO's would be:

PUTF	CH, E	Put flag
PUTD	CH, E	Put data
PUT	CH, E	Put record
GETFD	CH, E	Get flag or data
GET	CH, E	Get record

In the above, "CH" is the logical channel number. The customary

OPEN or INIT UUO is used to open the channel. Standard format user buffers are assigned. However, the ring and buffer headers will be used in a nonstandard way, so that data mode 12 is assigned for use with Network buffering and file status bit 31 must be on for output. (Any of the devices DSK, DTA, MTA, or IMP can be used in this mode.)

In the Harvard NCP and HOST-HOST protocol implementation, user buffers do not correspond directly to messages. On output, each user buffer will be formatted into a message; on input, a message may become one or two user buffer loads (128 word buffers are used in order to make maximum use of the facilities of the disk service routines).

PUTF UUO:

This UUO places a flag into the output buffer. The effective address is the location of a word:

XWD operation, count

In the case of block flags, the count is ignored, since it will be computed from the number of bytes actually placed in the buffer before the next use of PUTF. PUTF and PUTD will insert EOM flags automatically as each buffer becomes full; if data bytes are currently being placed in the buffer by PUTD, it will also insert an EOM flag after computing the count for the previous block flag in the buffer and place a new block flag of the same type at the beginning of the next buffer, after inserting a SIZE flag stating the then current byte size.

PUTD UUO:

This UUO places data into the output buffer. The effective address is the location of the data byte (if the byte size is less than 36) or of the next 36 bit word of data to be placed in the buffer. In the first case, the byte is assumed to be in the low order part of the word addressed. In the second case, the data word containing the final bits of the byte contains them in the high order part of the word, and the next data byte starts a new word in PDP-10 storage. Thus, for a byte size of 64, two entries to PUTD would be used per byte transmitted, the first containing 36 bits and the second containing 28 bits, left-justified. This strategy allows maximum use of the PDP-10 byte handling instructions.

PUT UUO:

This UUO places a whole logical record in the output buffer(s). The effective address is that of a word:

IOWD count, location

A PUTF UUO must have been used to output the proper SIZE flag. Thereafter, each use of PUT will output a BLOCK flag,* simulate a number of calls to PUTD using the IOWD to discover the location and size of the user data area, and then output a RS flag to indicate end of record.

---

*PUTF may be used to output the block flag, if a flag different from BLOCK is required.

In the case of byte size of less than 36 bits, PUT will use the ILDB instruction to pick up bytes to be output by PUTD. Hence, the standard PDP-10 byte-handling format is used, and the count part of the IOWD is the total byte count, not word count.

The above UUO's have both an error return and a normal return.

GETFD UUO:

The calling sequence for this UUO is:

```
GETFD CH, E
error return
whyte flag return
block flag return
data return
```

The effective address is the location at which the flag or data will be returned. The flag is returned in the same format as for PUTF and the data in the same format as for PUTD. Certain flags (NUL, IGNORE, and EOM) will be handled entirely within the UUO and will not be reported to the user. SYS should eventually be handled this way, but initially will be handled by the user.

GET UUO:

The calling sequence for this UUO is:

GET CH, E

error return

end of file return

end of group return

normal return

GET transmits the next logical record to the user, using GETFD together with an IOWD in the same format as for PUT. If the IOWD count runs out before end of record, the remainder of the record will be skipped. In any case, the updated IOWD will be returned at the effective address of the UUO in order to inform the user how much data was transmitted or skipped.

PDP-10 FILE TRANSMISSION:

Assume that I have a link connected to another PDP-10 and a user process there that is listening. In order to get that process to send me a file, the sequence of flags that might be transmitted can be represented as follows, where the UUO's executed by me are in the left margin, the flags are indented, and the commentary opposite them indicates the nature of the data block transmitted:

PUTF

CONTROL	Data with OPEN parameters, requesting OPEN
LABEL	File identification data for LOOKUP
EOM	Forces message to be transmitted

GETFD

STATUS	Status returned by OPEN

SIZE Byte size to be used

LABEL File retrieval information

PUTF

CONTROL Data requesting INPUT from file

EOM Forces request to be transmitted

GETFD

STATUS Status bits returned by INPUT

GET Logical record (one file buffer load)

(loop back to second PUTF, above, for other records)

Finally, the status information returned by the second GETF indicates end of file, and I wind up with the sequence:

PUTF

CONTROL Data requesting a CLOSE

EOM Forces transmission

GETFD

STATUS Status bits returned by CLOSE

In the case I am sending a file, the main loop looks like:

PUTF

CONTROL Data requesting OUTPUT

PUT Logical record (one file buffer load)

PUTF

EOM Forces transmission

GETFD

STATUS Status bits returned by OUTPUT

The use of both the record and the flag transmission UUO's is

worth noting, as well as the use of the EOM flag to force transmission of a message when switching between input and output over the link. PUT and GET UUO's are clearly required above for transmission of the CONTROL and LABEL data; I suppressed them for the sake of clarity.

For this application, the handshaking nature of the transmission of CONTROL and STATUS flags is mandatory. While the protocol would permit transmission of a complete file without the handshaking, it would be an all or nothing proposition -- a single error would necessitate doing it all over again, presuming that the receiving process did not end up in a complete tangle.

BRIEF DISCUSSION:

The PDP-10 space required to implement the above protocol is about 400 instructions, divided about equally between the input and the output side. Enough experimental coding has been done to confirm the feasibility of this basic strategy, taken together with experience with implementation and use of the SOS buffering system.

The above does not touch the question of LOGON protocol, except indirectly. My belief is that it can be accommodated in the framework of this proposal, but I have not tested this theory as yet. As indicated further above, I would be tempted to handle the matter with the SYS flag, given that SYS data is interpreted directly by the system (in our system, we would use the RUN UUO to run the LOGON CUSP, which would, in turn, handshake using

ASCII data over the link). In this way, I think we might be able to dispense with the notion of dedicated sockets and the reconnection morass.

One other point that needs thought is the question of how to handle the interrupt on link facility. Should it have any direct relation to the GET/PUT UUO's, or be handled on the side? I am inclined to think that it should be treated qua interrupt of the user process, quite independently of the matter of data transmission over the link. Some of our current work on the PDP-10 monitor would lend itself rather easily to implementation as a true interrupt.